KS12 Modem Technical Description

Project Description

The 1200 bps KISS modem software project was contrived from a need to get up to speed programming and using the TAPR/AMSAT DSP-93 platform. The practicality of a 1200 bps modem is questionable due to the availability of cheap TNC's and limited 1200 bps packet activity nowadays but this project has served as a useful learning tool and could be used for the basis of other more useful projects.

Some of the modem features:

- Bell 202 1200 Baud AFSK tone detection and generation.
- Performs HDLC Frame assembly and disassembly.
- Implements open squelch carrier detection.
- Communication with modem uses KISS protocol over the DSP-93 UART link.

By using some form of NOS(Network Operating System) on a PC, one does not need to use a TNC in order to connect to AX25 bbs's or run TCP/IP. All the high level protocol is implemented in the NOS software rather than inside a dedicated TNC. The communications between the PC and the DSP-93 take place over a standard RS-232 asynchronous serial line using a SLIP(Serial Line IP) derivative protocol commonly called KISS("Keep It Simple, Stupid"). This protocol specification is available widely on bbs's and the Internet so will not be delved into here.

Hardware Resources

The DSP-93 platform consists of a 40 MHz Texas Instruments TMC320C25 16 bit DSP chip, surrounded by 32K words of program memory and 32K words of data RAM. An analog interface board contains a TI TLC32044 14 bit A/D, D/A converter, a asynchronous UART chip, and various I/O ports for radio control and LED display control. A software controllable gain block is provided for adjusting the receiver input level to the A/D converter. A monitor EPROM is used to provide a downloader function as well as storing built in modem software and test applications. Programs can be downloaded into the DSP-93 using utility programs that run on a PC.

Software Design Method

The software for the KS12 modem was designed in a modular fashion using “C” language blocks to describe each function. Once the code blocks were defined, then the C320-25 assembly code was hand assembled using the “C” code blocks as a reference. This may seem cumbersome but designing in assembly language can get very complicated and confusing in a hurry even with generous comments. By designing the code in a higher level language, one doesn’t get bogged down in implementation details until the design is ironed out. This may take a little longer to get to the debug stage, but reduces the number of bugs once you get there, especially as the program gets more complicated.

The software source was also broken into several parts mainly to ease in editing. This sort of implements a “poor man’s” linking assembler in that one can edit and debug using just the file associated with a general task instead of having to search through one large cumbersome source file. When assembling of course, everything has to be re-assembled.

Data queues(FIFO, Circular, or “rubber band” buffers) are used throughout to reduce the timing constrains on the software and allow even distribution of processing time.

The only time critical operation is the actual A/D and D/A sampling operation which is performed by the TLC32044 CODEC chip in conjunction with the DSP hardware. Since the data is taken from or put into the data sample queues at precise time intervals, the rest of the software is not constrained to operate on the data in real time. This allows the software to perform periodic operations longer than the sample time interval as long as the average processing time does not exceed the sample time interval.

Another software method used was the use of indirect function calls to implement state machines which are of use at various places in the code. Basically the address of the function to call is placed in a RAM variable. An indirect call using that RAM variable results in execution of the specified function. Within that function, a “NEXT STATE” can be specified by simply loading the RAM variable with the address of the next state function. In this manner, complicated state machines can be implemented fairly easily.

The software is broken into eight files for easier manipulation:

- KS12MAIN.ASM This is the main entry file and is the one that is specified for assembling as it has all the include references for the auxiliary files. It contains all the Constant definitions, variable allocations, hardware and software initialization, interrupt service routines, and low level bit twiddling functions. The main code service loop also resides here which calls all the other modules in a round robin fashion to service all the various tasks of the modem.

- KS12DATA.TBL This file contains various constant data tables used throughout the modem. A SIN table, a CRC table, FIF coefficients , and various lookup tables are contained here.

- KS12AIN.ASM This file contains all the routines that service the A/D input samples as they arrive and demodulate, and decode the HDLC data into raw data bytes that are sent to the KS12HIN module.

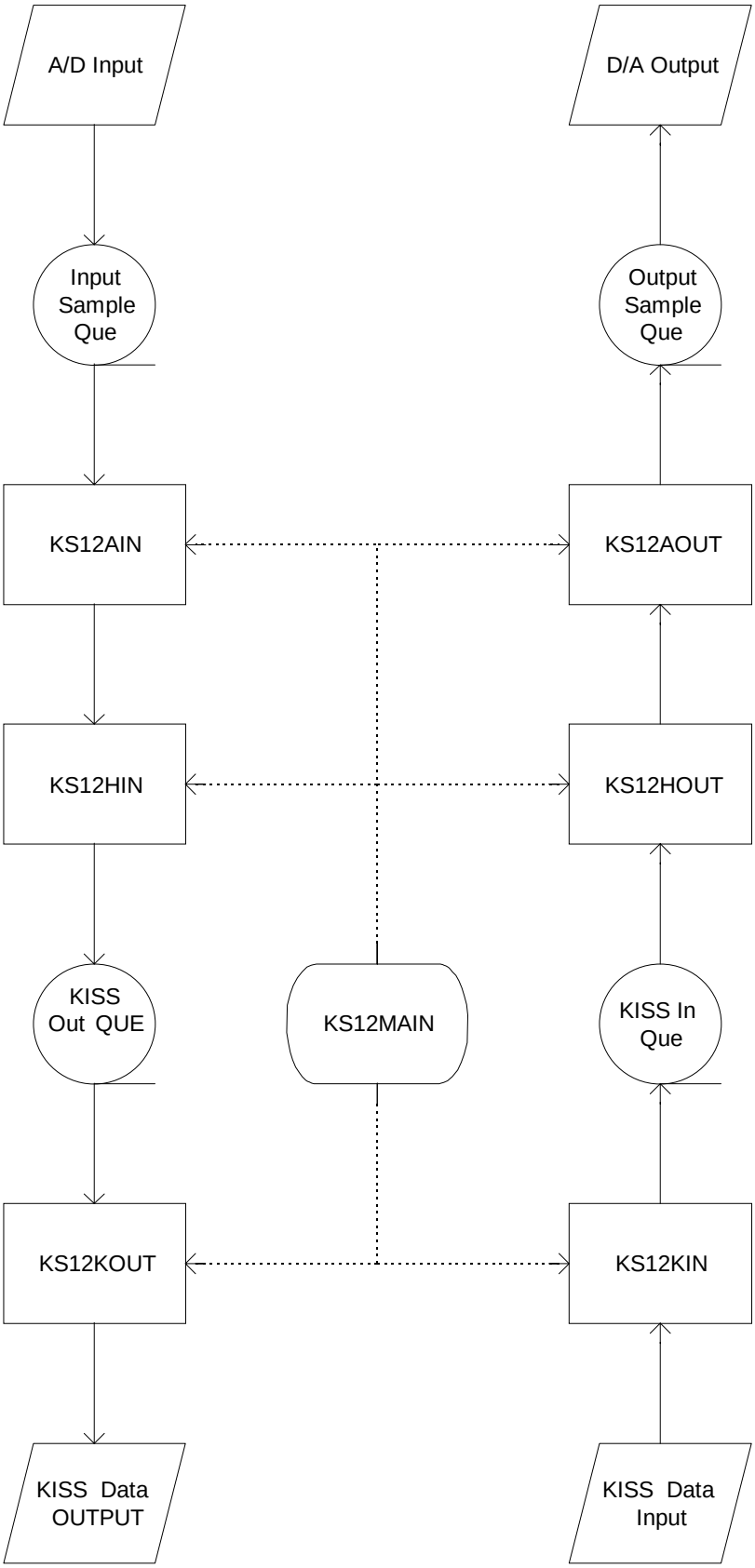
- KS12AOUT.ASM This file contains all the routines that create the Bell 202 AFSK tones in the proper NRZI HDLC sequence from an input byte of data that comes from the KS12HOUT module. The D/A output samples are generated here.

- KS12HIN.ASM This file contains all the routines that take the raw data bytes from the KS12AIN module and generate packets of data.

- KS12HOUT.ASM This file contains all the routines that take the data packets from the KS12KIN module and generates HDLC data bytes that are sent to the KS12AOUT module.

- KS12KIN.ASM This file contains all the routines that service the KISS formatted input bytes as they arrive from the DSP-93 UART and creates data packets for the KS12HOUT module.

- KS12KOUT.ASM This file contains all the routines that generate the KISS formatted output data from the data packets as they are created by the KS12HIN module.


A/D Input
D/A Output
Input Sample Que
Output Sample Que
KS12AIN
KS12AOUT
KS12HIN
KS12HOUT
KISS Out QUE
KS12MAIN
KISS In Que
KS12KOUT
KS12KIN
KISS Data OUTPUT
KISS Data Input

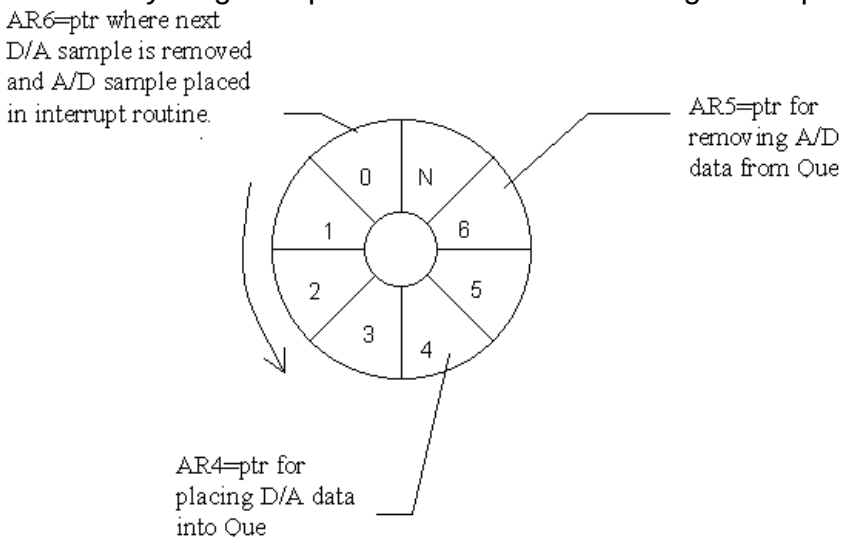
Software Descriptions

The software begins executing after being downloaded by first initializing the hardware resources on the DSP-93. The TLC32044 AIO chip is initialized to run at a sample rate of 10893 sample per second. The 16C550 UART chip is initialized to 9600 bps. This is the KISS interface rate(It can be changed by recompiling with a different constant). The onboard timer of the TMS320C25 chip is set to interrupt every 5 milliseconds. This is used as a general purpose timer for some of the initialization routines, LED display, and transmitter timing functions. Several initialization routines are called to initialize various variables used by each module. After initialization, the interrupts are enabled, and the main service loop is entered in which all seven software modules are called in a loop continuously.

First the AIO interrupt service routine will be described. It's function is to send a new sample from the Sample_Que out the D/A PORT and store a new A/D sample into the Sample_Que. Since the A/D and D/A are run at the same sample rate, only one interrupt service routine is used for both. Also since one word is removed and one word is placed in the Sample_Que at every sample time, only 3 pointers are need to maintain the Sample_Que.

```
void RxIntService(){
   Save_Context();
   DXR = Sample_Que[AR6];              // write D/A from Sample_Que
   Sample_Que[AR6++] = DRR;            // read A/D into Sample_Que
   if( AR6>Sample_Que+SAMPQUESIZE-1 )  // deal with wrap around
      AR6 = Sample_Que;
   Restore_Context();
}
```

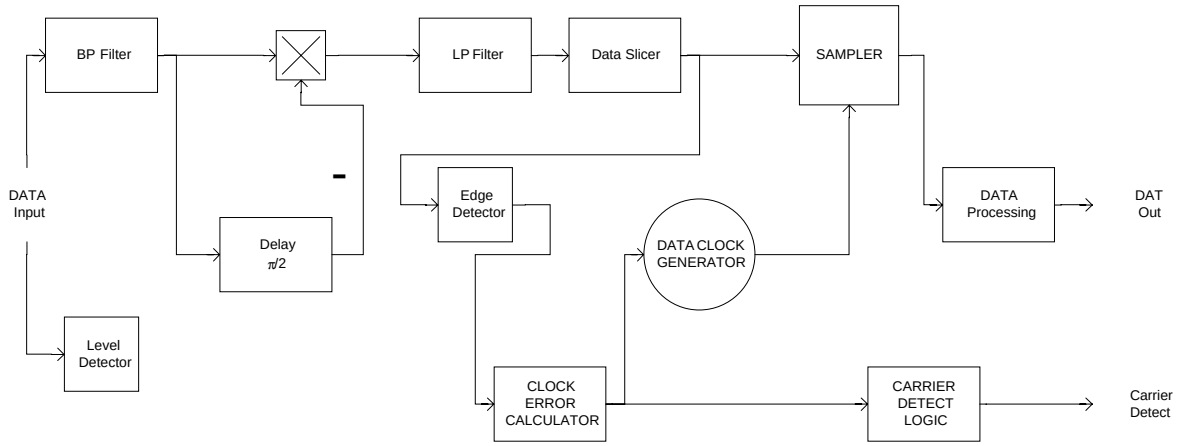
Three Auxiliary registers(AR6,AR5,AR4) are used as pointers to the Sample_Que. AR3 is used as a software stack pointer to save and restore processor context since the 320C25 doesn't save anything except the return address during interrupts.



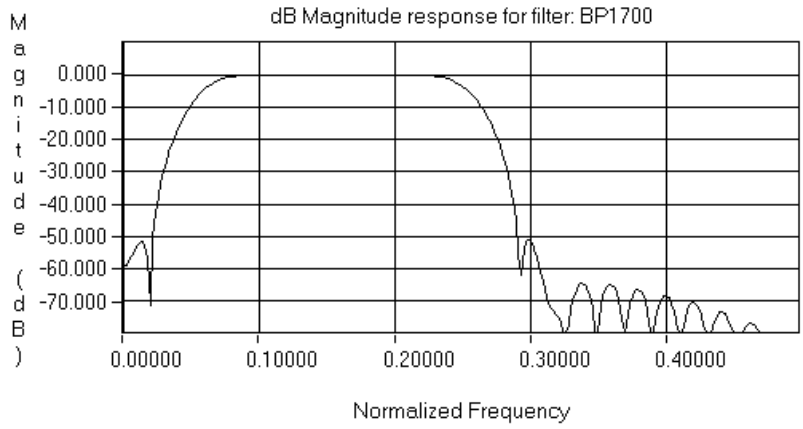
A/D data can be removed from the Sample_Que as long as AR5 != AR6.
D/A data can be placed in the Sample_Que as long as AR4 != AR5.

KS12AIN.ASM Module

The block diagram below shows the general operation of this module. A/D samples are pulled out of the Sample_Que and passed through a level detector. This block just sees if any samples are above some peak threshold and flashes a front panel LED. This is useful in setting the receive audio level. Next the data is passed through a Band Pass FIR filter that is wide enough to pass the AFSK signal.



The 50 bandpass filter coefficients were determined using a program called “PC-DSP” which takes filter parameters and creates a coefficient table as well as performs analysis on the filter. Below is the generated magnitude plot of the filter. The frequency is normalized to the sample frequency.(multiply by the sample frequency to get the actual frequency points)



Delay Line FSK Demodulator Derivation

After passing through the band pass filter, a delay line FSK demodulator is used to detect the tones from the incoming signal. The basic idea of the demodulator is to multiply the incoming signal by a negative delayed($\pi/2$ radians) version of itself. Passing this signal through a low pass filter to remove a 2X frequency component results in a signal whose sign is the digital data(1 or 0).

Where ω_c = carrier frequency = $2\pi1700$ rad/sec and $\delta\omega = 2\pi500$ rad/sec for Bell 202 tone frequencies.

$x(t) = \cos(\omega_c \pm \delta_\omega)t$ where x(t) is the binary modulated input signal

$x(t-\tau) = \cos[(\omega_c \pm \delta_\omega)(t-\tau)]$ where x(t-τ) = τ delayed version of x(t)

Multiplying together yields z(t)

z(t) = x(t)x(t-τ)

$z(t) = \cos[2(\omega_c \pm \delta_\omega)t - (\omega_c \pm \delta_\omega)\tau] + \cos[(\omega_c \pm \delta_\omega)\tau]$

If let $\omega_c\tau = \pi/2$ and the $2\omega_c$ frequency component is lowpass filtered out, then

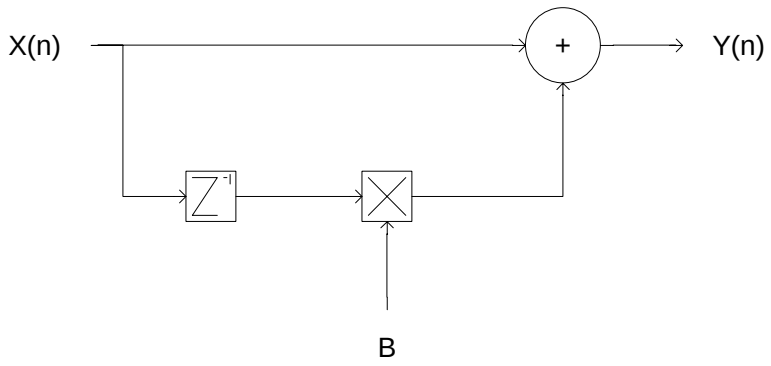
$z(t) = \cos(\pi/2 \pm \delta_\omega\tau) = \sin(\pm\delta_\omega\tau) = \pm\sin(\delta_\omega)$

For this modem:

$\omega_c\tau = \pi/2$ since $\omega_c = 2\pi f_c$ so $\tau = \frac{1}{4 f_c}$ = 147.0588μsec

The problem now is how to delay 147.0588μsec when the sample period is fixed at 1/10893= 91.802μsec. A delay of 1.6019 sample periods is needed. By delaying one sample that leaves a need to delay .6019 of a sample period. The answer is to implement a single zero filter of the form

Y(n) = X(n) + BX(n-1)



The transform of this filter is:

$H(z) = 1 + Bz^{-1}$

First find the frequency response of the filter by letting $z = e^{j\omega}$
where $\omega = 2\pi Fc/Fs$ and Fc is carrier frequency and Fs is the sample frequency

$H(\omega) = 1 + B\, e^{-j\omega}$

since $e^{-j\omega} = \cos(\omega) - j\sin(\omega)$

$H(\omega) = 1 + B\cos(\omega) - jB\sin(\omega)$

The phase response of this filter is

$\phi(\omega) = \arctan(- B\sin(\omega) / (1 + B\cos(\omega)))$

the group delay is the derivative of the phase response so:

$$\tau = \frac{-d\Phi(\omega)}{d\omega} = \frac{-d}{d\omega}(\arctan \frac{-B\sin(\omega)}{+B\cos\ \omega}$$

where τ = delay expressed in terms of sample delays.

Going through the gory details yields a solution for the value B given a frequency ω and a desired delay τ.

$$B = \frac{(2\tau - 1)\cos(\omega) \pm \sqrt{(1-2\tau)^2 \cos^2(\omega) + 4\tau(1-\tau)}}{2(1-\tau)}$$

where:
$\omega = 2\pi Fc/Fs$ (Fc is carrier frequency and Fs is the sample frequency) and
τ = desired group delay in fraction of sample time.

For this modem, Fs = 10893, Fc=1700, and the desired fractional delay is .6019

Solving for B gives:

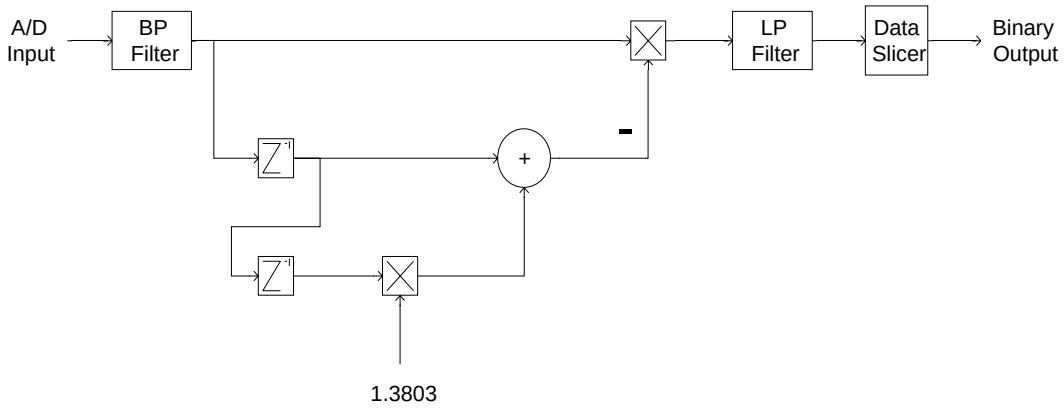
B=1.380287
A “C” program to calculate this is given below:

```
/*=========================================================================*/
/*                      F R A C T P H Z . C                                */
/* Program to calculate fractional phase delay coefficient for one stage   */
/* zero filter given sample freq and input frequency.                      */
/*=========================================================================*/
/*-------------------------------------------------------------------------*/
/*------------------------> I N C L U D E S <-----------------------------*/
/*-------------------------------------------------------------------------*/
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <math.h>

#define PI 3.14159
double Fs, Fi;
double Beta, D, A;
double w;
void main( void );

void main( void )
{
double Num, Den, sqrtval;
   printf( "\nEnter Fractional sample delay desired(0-.999) ->");
   scanf("%lf", &D );
   printf( "\nEnter Input frequency in Hz ->");
   scanf("%lf", &Fi );
   printf( "\nEnter Sample frequency in Hz ->");
   scanf("%lf", &Fs );

   printf("\n Desired Delay=%lf ",D );
   printf("\n Desired Fin=%lf ",Fi );
   printf("\n Desired Fsamp=%lf ",Fs );

   w = (Fi/Fs)*2.0*PI;
   Den = 2.0*(1.0-D);
   sqrtval = (1.0-2.0*D)*(1.0-2.0*D)*cos(w)*cos(w) + 4.0*D*(1.0-D);
   if( sqrtval >= 0.0 ){
      Num =  ( (2.0*D-1.0) * cos(w) ) - sqrt( sqrtval );
      Beta = Num/Den;
      printf("\n Beta1=%lf ",Beta );
      A = sqrt( (1.0+Beta*cos(w))*(1.0+Beta*cos(w)) +
                              Beta*Beta*sin(w)*sin(w) );
      printf("\n Gain=%lf ",A );
      Num =  ( (2.0*D-1.0) * cos(w) ) + sqrt( sqrtval );
      Beta = Num/Den;
      printf("\n Beta2=%lf ",Beta );
      A = sqrt( (1.0+Beta*cos(w))*(1.0+Beta*cos(w)) +
                              Beta*Beta*sin(w)*sin(w) );
      printf("\n Gain=%lf ",A );
   }
   else
      printf( "Improper input values\n" );
      exit(0);
}
/*====================   F R A C T P H Z . C   ==========================*/
```

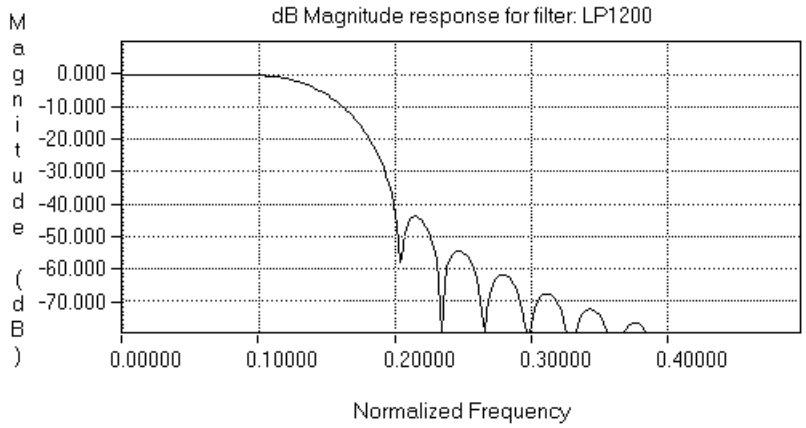
Program output:

```
Desired Delay=0.601900
Desired Fin=1700.000000
Desired Fsamp=10893.000000
Beta1=-1.095375
Gain=0.990250
Beta2=1.380287
Gain=2.107505
```

Putting all this together yields to following implementation of the phase delay FSK demodulator:



The low pass filter is a 32 tap FIR filter that is needed to remove the 2X frequency components generated by the multiplication process. It was designed using “PC-DSP”. Below is the generated magnitude plot of the filter. The frequency is normalized to the sample frequency.(multiply by the sample frequency to get the actual frequency points)



The sign of the LP filter output is the recovered binary data stream which now can be further processed.

NRZI FORMAT

The binary data stream is encoded in the following manner:

The raw data to and from the modem is encoded in NRZI format (Non Return to Zero Inverted). NRZI data basically is a '1' if the present bit is the same as the previous bit and '0' if the present bit changed from the previous bit.

Example data stream:

```
raw recv data =   0   1   0   1   0   1   1   1   1   0   1   0   0   0
actual data   =   x   0   0   0   0   0   1   1   1   0   0   0   1   1
```

Whenever the input data changes, the actual data is a zero. If the input data is the same, then the actual data is a one.

The NRZI data provides an interesting feature in that it really doesn't matter what the polarity of the incoming data bits is since the data comes from the changes in polarity.

Example of same data stream only inverted:

```
raw recv data =   1   0   1   0   1   0   0   0   0   1   0   1   1   1
actual data   =   x   0   0   0   0   0   1   1   1   0   0   0   1   1
```

Note the received data is the same.

ZERO BIT INSERTION

Since there is no clock associated with this data stream, the bit positions need to be extracted from the data itself. One way is to look at the time when data transitions occur then extrapolate where the ideal center of the bit should be. One problem is when a long stream of zeros or ones occurs in the incoming data. Since there are no transitions, it would be hard to determine the bit positions. The HDLC format does what is called zero bit insertion (or bit stuffing) to insure that transitions in the NRZI data stream occur frequently enough to be able to determine bit position.

Bit stuffing basically limits the number of contiguous ones to 5 bits. If 5 one bits occur in a row, an extra zero is inserted into the data stream. Note we are only concerned with ones because consecutive zeros in the data stream get converted to data transitions due to the NRZI encoding. Long strings of data zeros create alternating ones and zeros due to the NRZI format in the outgoing data stream and so are not a problem.

Example of bit stuffing (transmitting data):

```
data to be sent  =   0   1   1   1   1   1     1   1   1   0   0   0
bitstuffed data  =   0   1   1   1   1   1  0  1   1   1   0   0   0
                           inserted zero bit^

NRZI data sent   =   0   0   0   0   0   0  1  1   1   1   0   1   0
   or
NRZI data sent   =   1   1   1   1   1   1  0  0   0   0   1   0   1
```

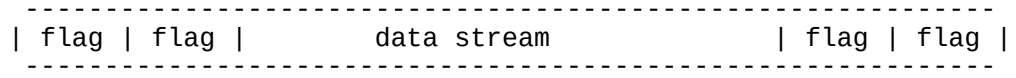
FLAG BYTE

Now that we have a stream of data with lots of transitions to determine the bit positions, how does one find the start and end of a string of data? HDLC provides what is called a flag character which is unique in that it cannot occur anywhere in the regular data stream. A flag byte is sent by sending a zero followed by 6 ones followed by another zero(7Eh). This byte is an exception to the 5 ones in a row bitstuffing rule and is why it can not occur in the normal data stream.

Each Frame(packet) of data starts with one(or more) flag bytes and after all the data is sent, one(or more) flags bytes is sent. In this way, it is possible to determine the beginning and end of each packet of data. If the decoder ever receives a zero followed by 6 ones followed by another zero(7Eh), then the decoder knows that a new frame is starting.(or is finished if data was being received prior to the flag)

Multiple Flags can and usually are sent at the beginning and/or end of a packet to give the decoder time to determine the center of the incoming data bits before the actual data starts.

FRAME(packet) STRUCTURE




```
flag byte = 0 1 1 1 1 1 1 0
```

example of a string of flags:

```
                              flag            flag            flag
multiple flag bytes =  0 1 1 1 1 1 1 0 0 1 1 1 1 1 1 0 0 1 1 1 1 1 1 0

NRZI flag bytes     =  1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 0
      or
NRZI flag bytes     =  0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1
```

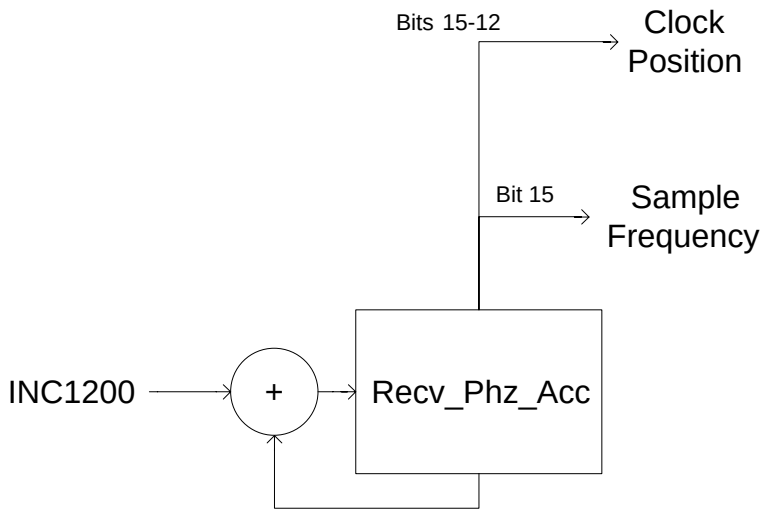
All the data bytes are composed of 8 bit bytes that are transmitted least significant bit first. (the last 16 bits is a CRC word which is transmitted msbit first)

Clock Recovery

The first thing that must be done to recover the data is recover the clock position for sampling the incoming data stream. This is done by implementing a free running 1200bps sample clock and phase locking it to the bit center position using the time data edge transitions occur to shift the sample clock phase until the data sampling occurs half a sample clock from any edge.

A 16 bit phase accumulator(Recv_Phz_Acc) is used to generate the sample clock. The phase increment value is chosen so that the word rolls over at a 1200 bps rate. The increment value (INC1200) that must be added every sample time is:

$$INC1200 = \frac{Fout(2^{16})}{Fs} = \frac{1200(2^{16})}{10893} = 7220$$



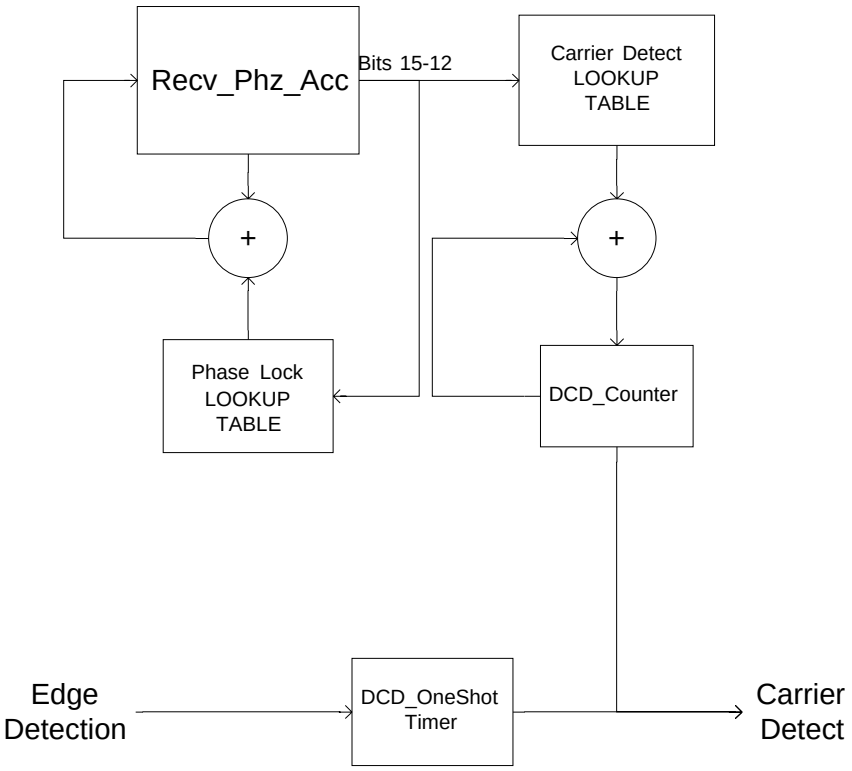
The upper 4 bits of the Recv_Phz_Acc are used determine clock position. It divides each bit time into 16 time slots. The actual data sampling occurs when these four bits are ZERO. (Actually when the counter rolls over from a negative to positive number. Due to the increment rate, some values may occur twice so it is better to just look at the rollover condition)

The routine Phz_Lock() is called at every data transition time. Since the center sampling point was chosen to be count 0x0 (upper four bits of Recv_Phz_Acc) the data edges should occur ideally halfway between count 0x0 and 0xF. By the use of a lookup table, indexed into by an average value of Recv_Phz_Acc, an error value can be determined based on the actual count at which an edge occurred. This error value is added to Recv_Phz_Acc in order to shift the sample position into the correct bit position.

Another function that can be obtained from this clock position error is whether a valid input signal is being received or just receiver noise. If noise is being received, the data edge positions will occur randomly and the resulting position error will also be random. By using another lookup table and an integrator counter(DCD_Counter), a carrier detect function can be implemented. If the input signal is valid, the clock position should be confined to a narrow range of values in the middle of the bit. The lookup table values are all positive in this region and are added to the DCD_Counter. When the DCD_Counter reaches a threshold limit, valid carrier detect is asserted and data decoding can begin. If noise is received, edge positions are distributed over the entire 0x0 to 0xF range. The lookup table has negative entries outside the nominal clock position range and so when added to the DCD_Counter, will cause it to decrement to

ZERO and turn off the carrier detect state. DCD_Counter is clamped to ZERO and a maximum count to prevent under and over flow.
Two separate lookup tables are used to determine carrier presence. A fast acting one is used before a valid signal is found. Once a carrier detect is asserted, a slower acting one is applied to help ride through noise bursts.

A timeout timer DCD_OneShot is loaded on every data edge. It is decremented at every sample time. If this timer ever decrements to ZERO, then the carrier detect is turned off. This is used to prevent the case where a valid carrier detect is asserted and then no more data edges are detected(such as if a squelched receiver is used or a unmodulated signal is received). Since no edges are occurring the normal carrier detect counter never gets serviced and the carrier detect would always be asserted.



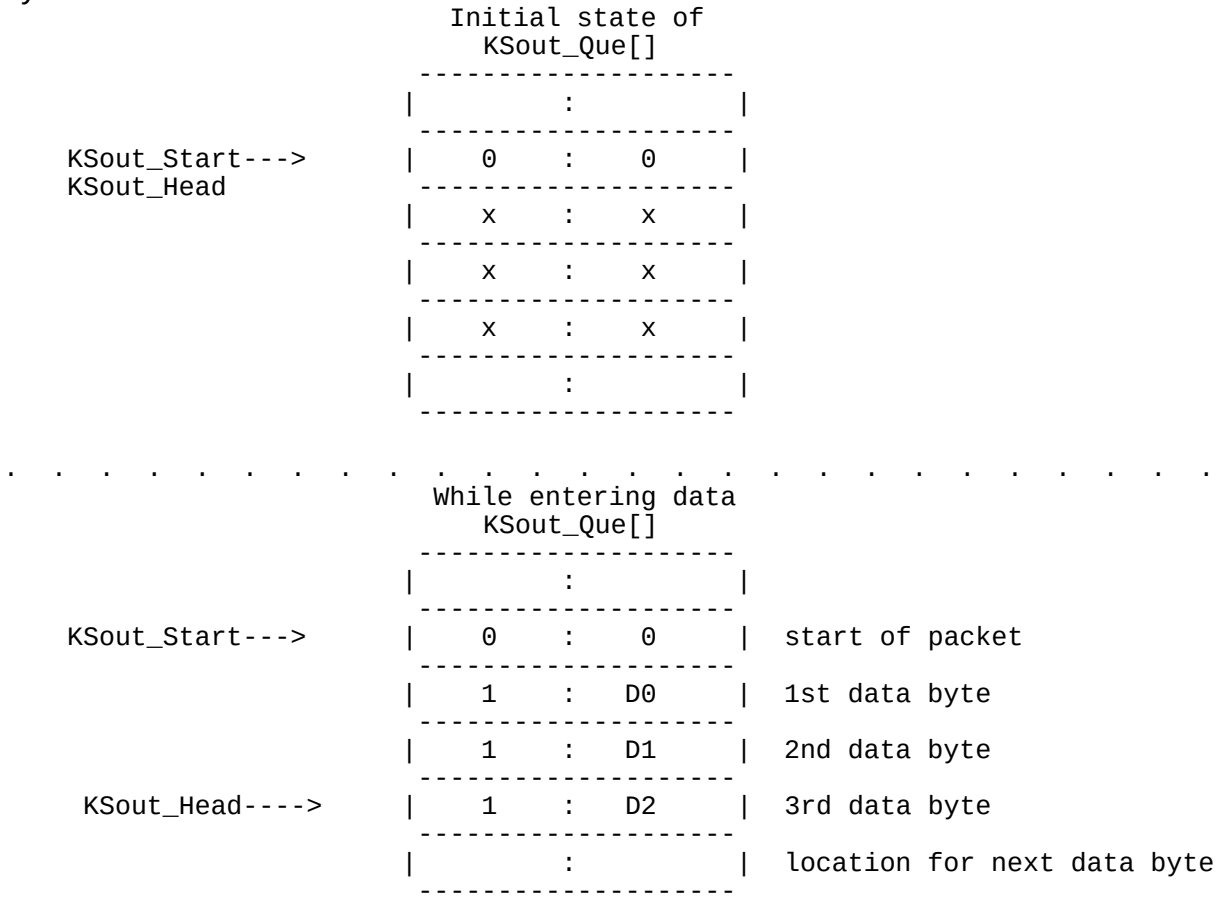
The last function in KS12AIN.ASM to be discussed is the routine Process_Inbits() which is called at every data sample time to assemble the incoming binary data stream into bytes of data and flag information that will be passed along to another module for further processing. The “C” code is shown below and basically converts the NRZI data into data and flag bytes removing any “bitstuffed” zero’s along the way. The variable FLAG.xxx is a reference to a BOOLEAN bit variable.

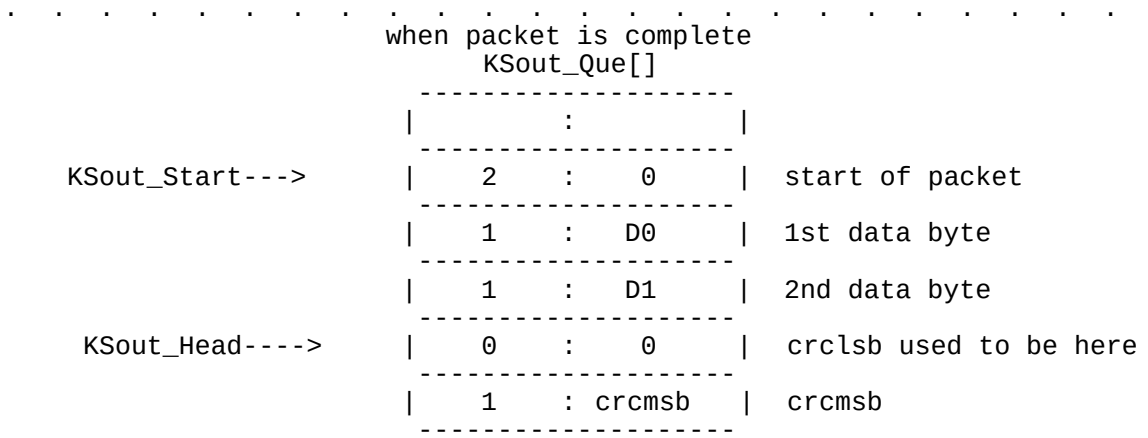
```
void Process_Inbits( Flags )
   if( Flags.CARRIERDET ){            // if carrier present
      if( Flags.RXLASTBIT == Flags.INPUTBIT ){
         if( RX_OnesCounter < 5 ){    // Input Bit is same as last so is ONE
            RX_OnesCounter++;         // shift in valid ONE data bit
            RX_ByteRegister >>= 1;
            RX_ByteRegister |= 80h;
            if( ++RX_BitCounter >= BITSINBYTE ){  // got a whole byte?
               RX_BitCounter = 0;
               RX_NewByte = RX_ByteRegister;
               Flags.NEWBYTERDY = TRUE;
            }
         }
         else{
            if( RX_OnesCounter < MAX_ONES ){ // clamp ones_counter to MAX
               RX_OnesCounter++;
            }
         }
      }
      else{            // Input bit changed polarity so is a ZERO data bit
         Flags.RXLASTBIT = Flags.INPUTBIT;
         if( RX_OnesCounter < 5 ){    // is not a flag or stuffed bit yet
            RX_ByteRegister >>= 1;    // shift in a zero
            RX_OnesCounter = 0;
            if( ++RX_BitCounter >= BITSINBYTE ){  // got a whole byte?
               RX_BitCounter = 0;
               RX_NewByte = RX_ByteRegister;
               Flags.NEWBYTERDY = TRUE;
            }
         }
         else{
            if( RX_OnesCounter == 6 ){   // is a flag so mark it as such
               RX_BitCounter = 0;
               RX_NewByte = HDLC_FLAG;
               Flags.NEWFRAMERDY = TRUE;
               Flags.NEWBYTERDY = TRUE;
            }                 // else is a stuffed bit so ignore this zero
            RX_OnesCounter = 0;
         }
      }
   }
   else{                               // here if carrier goes away
      Flags.NEWBYTERDY = TRUE;         // reset receiver variables
      Flags.NEWFLAGRDY = TRUE;
      RX_BitCounter = 0;
      RX_OnesCounter = 0;
      RX_ByteRegister = 0;
   }
}
```

KS12HIN.ASM Module

This module takes the data bytes received by the KS12AIN module and separates them into valid data packets and places the packet into a data queue “KSout_Que”. The code waits for a the first HDLC flag byte to be received signaling the beginning of a new packet. Further flags are ignored until the first data byte is received. Data bytes are then placed in the output queue until another flag byte is received marking the end of the packet, or carrier detect is lost. The data is validated by checking the CRC of the packet and if good, the assembled data packet is marked as valid for conversion and transmission out the KISS UART port.

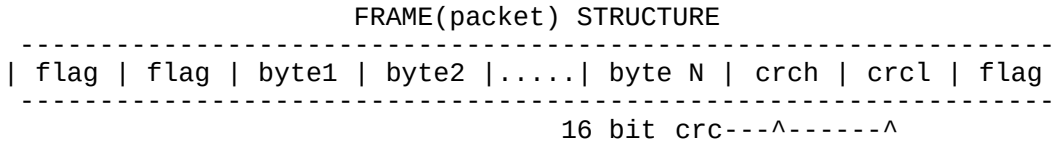
Since data is in the form of 8 bit bytes and the data queues are 16 bit words, the upper byte of the word can be used to mark the data positions within the data queue as ready or not ready. This is useful because one doesn’t know until all the data bytes are received whether or not the packet is good because the CRC check is done after all the bytes are received.





CALCULATING FRAME CHECK BYTES

The last two bytes in every frame(packet) of data consist of a 16 bit crc(cyclic redundancy code) to be able to check the integrity of the packet. It is calculated on all the bytes in the Frame excluding the flag bytes. The calculation is based on ISO 3309 recommendation using a generator polynomial(divisor) of $x^{16} + x^{12} + x^5 + 1$. The 16 bit crc is sent most significant bit first.



When receiving a packet, the crc is calculated on the incoming data bytes as they come in. When the ending flag byte is received, the calculated crc from all the incoming data bytes(including the incoming crc) should equal 0xF0B8. If it doesn't then the frame should be discarded since it has one or more data errors.

The derivation of the generation of crc's is beyond the scope of this text but an algorithm using a table look-up method is described. It uses a pre-calculated 256 WORD look up table. The following code generates the 256 word crc lookup table:

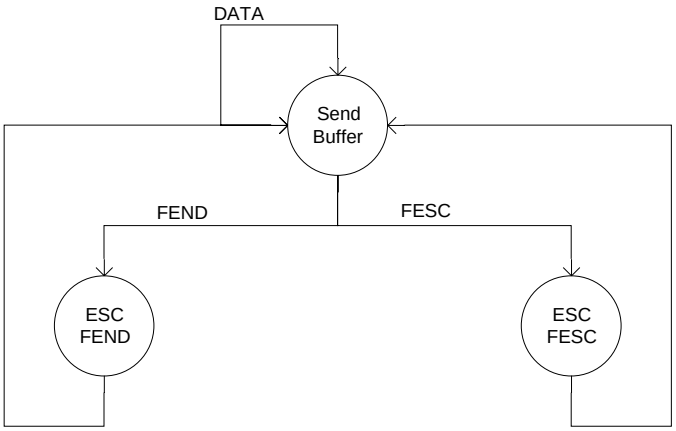
```
//==============  CRC table generating code  =================
//This program will generate the 256 word table for use in calculating
//crc's
//
const int MagicNums[8] = {
   0x1189, 0x2312, 0x4624, 0x8C48, 0x1081, 0x2102, 0x4204, 0x8408 };

void main()
{
int i,j;
unsigned short value;
   printf("CRC_TABLE:              ;CRC generation table\n");
   for(i=0; i < 256 ; i++){
      value = 0;
      for(j = 0; j<8; j++)
         if( i & (1<<j) )
            value ^= MagicNums[j];
      printf("        .word   0%4.4Xh  ;%u\n",value, value);
   }
   exit(0);
}
//==============  CRC table generating code  output================
CRC_TABLE:              ;CRC generation table
        .word   00000h  ;0
        .word   01189h  ;4489
        .word   02312h  ;8978
        .word   0329Bh  ;12955
        .word   04624h  ;17956
        .word   057ADh  ;22445
            ...
            ...
            ...

        .word   02C6Ah  ;11370
        .word   01EF1h  ;7921
        .word   00F78h  ;3960
```

The following routine is used to calculate the 16 bit crc.

```
//==============  CRC generating code  =================
// To use, first initialize crc to -1 (0xFFFF).  Then call the
// routine Calc_crc( x ) for every new data byte 'x'.
// When finished, the new crc value is in 'crc'.

int crc, crc_Temp;
void Calc_crc( unsigned char newbyte )
{
   crc_Temp = newbyte ^ crc & 0xFF;
   crc >>= 8;
   crc &= 0x00FF;
   crc ^= CRC_TABLE[ crc_Temp ];
}
```

KS12KOUT.ASM Module

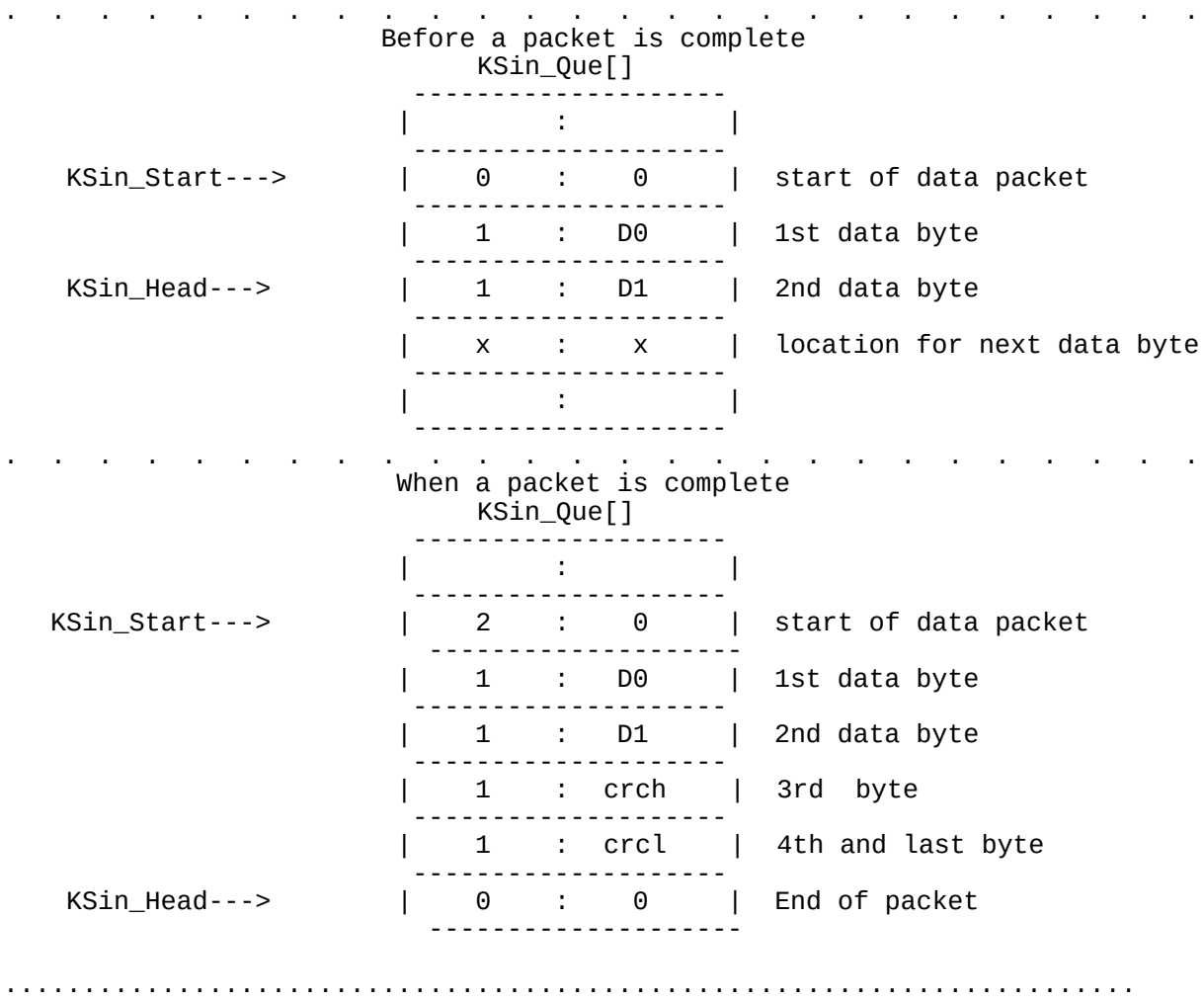
This module waits until a completed packet of data is received in the Ksout_Que and then sends it out the UART port using the KISS protocol format. About the only exciting thing this module does is implement a state machine that helps in creating the proper FEND frame separators and also the escape sequences needed in the protocol.



KS12KIN.ASM Module

This module waits for a KISS format data byte to be received by the UART and places it in the circular buffer "KSin_Que" if it is KISS data or decodes the KISS command. The incoming data packet is placed in the KSIn_Que which is configured similarly to the queue KSOut_Que in that the upper 8 bits of each data word contain information on whether the data is complete and also separates multiple data packets.

```
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
                          Initial state
                            KSin_Que[]
                         --------------------
                        |         :          |
                         --------------------
    KSin_Start--->      |    0    :    0     |  start of data packet
    KSin_Head            --------------------
                        |    x    :    x     |
                         --------------------
                        |    x    :    x     |
                         --------------------
                        |    x    :    x     |
                         --------------------
                        |         :          |
                         --------------------
```



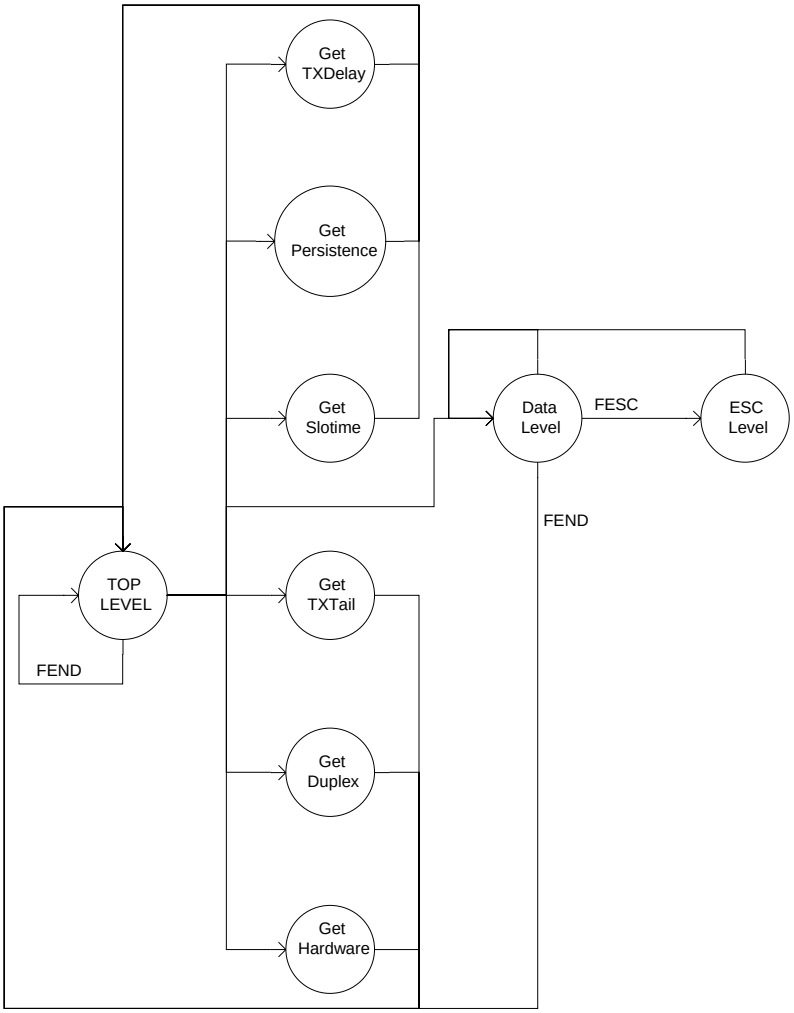
After a KISS data packet is received a CRC word is generated and placed at the end of the data. A duplicate of the receive crc calculation routine is used in order to allow duplex operation where incoming and outgoing packets require crc calculations. The same look up table is used for both.

KISS commands which are identified by a non zero byte following the FEND byte are decoded and the various parameters are stored in RAM variables for use by the system. The KISS parameters are described in detail in the KISS specification. The only unique parameter not specified by the KISS specification is the Hardware parameter byte. For this modem it is used to allow the user to change receiver gain settings and also choose radio port 1 or 2 using the KISS connection without re-assembling the code. The following table describes the allowable values for the Hardware parameter byte:

Bit 7 selects radio port1 or 2. (default is port1, Bit7=0) Bits 5-0 select the Receive gain.

param DEC	param HEX	Radio Port	Receiver Gain
0	00h	1	Gain = 1
8	08h	1	Gain = 2
16	***10h***	***1***	***Gain = 4***
24	18h	1	Gain = 8
32	20h	1	Gain = 16
40	28h	1	Gain = 32
48	30h	1	Gain = 64
128	80h	2	Gain = 1
136	88h	2	Gain = 2
144	90h	2	Gain = 4
152	98h	2	Gain = 8
160	A0h	2	Gain = 16
168	A8h	2	Gain = 32
176	B0h	2	Gain = 64

Default settings are Radio Port1 with a GAIN=4.



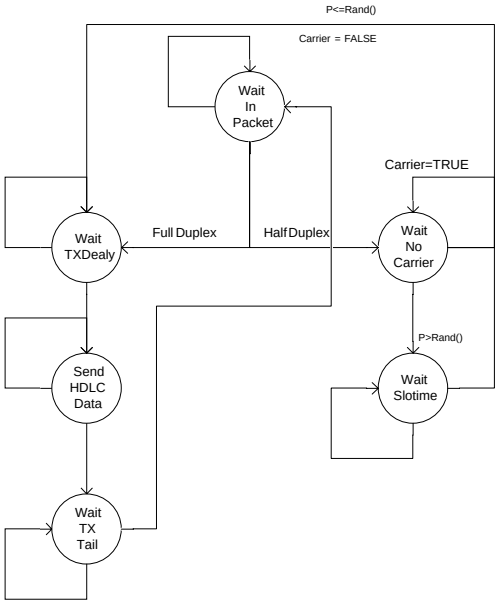
KS12KIN.ASM State Machine

KS12HOUT.ASM Module

This module waits for a complete data packet to be placed into the KSIn_Que and then begins the process of transmitting the packet. If the Duplex mode is activated, the transmitter is keyed and a string of flag bytes is sent to the KS12AOUT.ASM module for conversion into the proper tone sequences. After a time delay of .01*TXDelay seconds, the packet data is removed from the KSIn_Que one byte at a time and sent to the module KS12AOUT.ASM. After the data is sent, flag bytes are sent for .01*TXTail seconds and then the transmitter is unkeyed.

If the half duplex mode is selected, the above sequence is followed except that the transmitter is not keyed until Carrier detect is inactive. Whenever data is ready for transmission, the process begins monitoring the carrier detect bit. It waits indefinitely for this signal to go inactive. When the channel clears, a random number between 0 and 255 is generated. If this number is less than or equal to the parameter “Persistence”, the transmitter is keyed on and the process waits .01*TXDelay seconds, then transmits all queued frames. After the data is sent, flag bytes are sent for .01*TXTail seconds and then the transmitter is unkeyed. If the random number is greater than Persistence, the process delays .01*SlotTime seconds and repeats the procedure beginning with the sampling of the carrier detect bit. (If the carrier detect signal has gone active in the meantime, the process again waits for it to clear before continuing). Note that Persistence = 255 means ``transmit as soon as the channel clears''.

The Persistence and SlotTime parameters are used to implement p-persistent CSMA.(Carrier Sense Multiple Access) This method can reduce the chance of collisions on an active channel. If several stations are waiting for the channel to clear, it is quite likely more than one will begin transmitting at the same time and interfere. By having each station wait a random period of time before transmitting, the likelihood of collisions is reduced.



KS12AOUT.ASM Module

This module takes bytes of data to be transmitted, serializes them, encodes the serial data in NRZI format, performs bitstuffing, clocks out the data at 1200BPS rate, and generates the proper D/A samples to create AFSK tones.

A 1200 BPS transmit clock is generated using a phase accumulator word that is incremented by INC1200 every sample time. A 16 bit phase accumulator (Xmit_Phz_Acc) is used to generate the sample clock. The phase increment value is chosen so that the word rolls over at a 1200 bps rate. The increment value (INC1200) that must be added every sample time is:

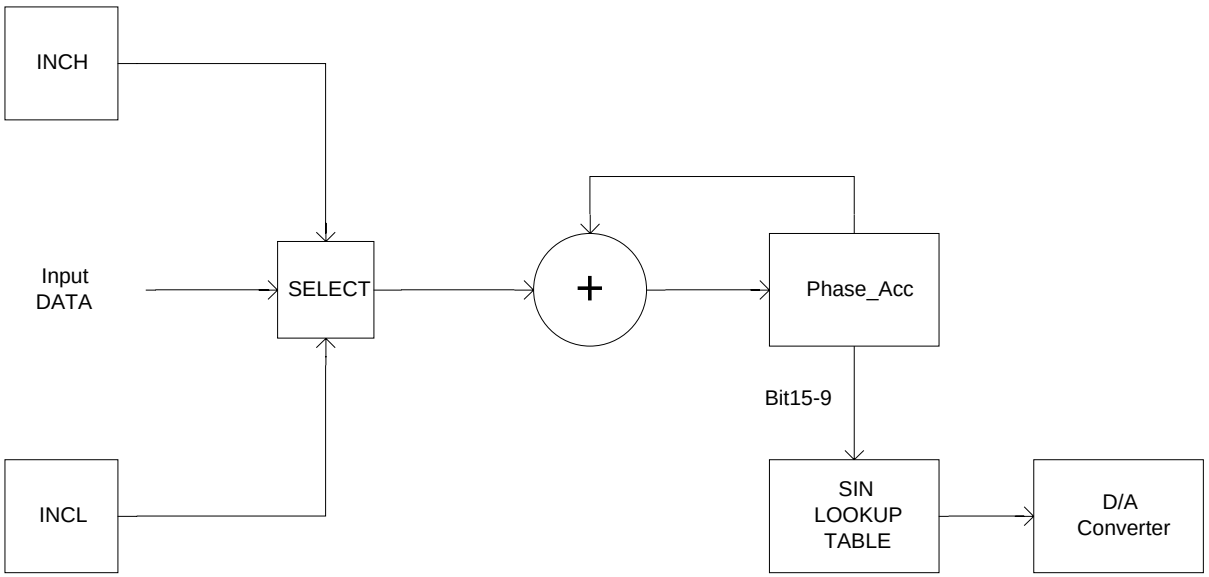
$$INC1200 = \frac{Fout(2^{16})}{Fs} = \frac{1200(2^{16})}{10893} = 7220$$

The routine BitTransmit() is called every time Xmit_Phz_Acc rolls over. This routine creates flag byte sequences and serializes the data bytes that are to be sent using bitstuffing and NRZI encoding. The final output bit value then is used to select which tone frequency to transmit. The “C” code is shown below:

```
void BitTransmit()
{
   if( Flags.TRANSMITON ){
      if( (TX_ByteRegister & 1) == 1 ){
         TX_OnesCounter++;
         if( TX_OnesCounter != 5 ){
            if( TX_OnesCounter > 5 ){  // see if need to ZERO insert
               TX_OnesCounter = 0;
               Flags.TXLASTBIT = ~Flags.TXLASTBIT;
               Flags.TXPRESENTBIT = Flags.TXLASTBIT;
            }
            nextbit();
         }
      }
      else{                            // if data bit is a ZERO
         Flags.TXLASTBIT = ~Flags.TXLASTBIT;
         Flags.TXPRESENTBIT = Flags.TXLASTBIT;
         if( Flags.FLAGING ){          // allow 6 ONE bits to be sent
            TX_OnesCounter = -5;
            Flags.FLAGING = FALSE;
         }
         else
            TX_OnesCounter = 0;
         }
         nextbit();
      }
      if( Flags.PRESENTBIT )
         PhaseInc = INCH;              // set 2200 Hz tone
      else
         PhaseInc = INCL;              // set 1200 Hz tone
   }
   else
      PhaseInc = 0;                    // turn off tone
}
```

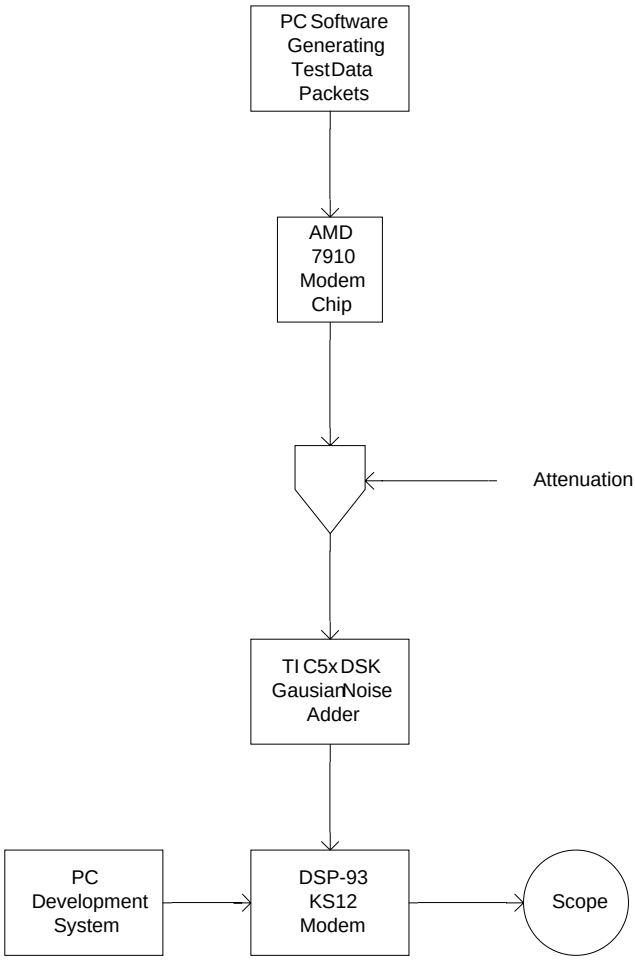
```
void nextbit()
{
   TX_ByteRegister >>= 1;
   if( ++TX_BitCounter >= BITSINBYTE ){
      TX_BitCounter = 0;
      if( Flags.SENDFLAG )
         Flags.FLAGING = TRUE;
      Flags.SENDFLAG = FALSE;
      TX_ByteRegister = TX_NewByte;
      Flags.NEXTBYTE = TRUE;   //signal ok to load next byte
   }
}
```

The output tones are created with yet another phase accumulator “PhaseAcc” that is incremented by the contents of a variable “PhaseInc” at the sample rate of the D/A. The upper 7 bits of the Phase Accumulator are used to index into a SIN lookup table to generate the output tones. By selecting one of two possible phase increment values(INCL,INCH), the two tone frequencies(1200,2200) required for transmission can be generated. When the transmitter is off, the increment value is set to zero so no tone is generated.



MODEM TEST METHOD

Several methods were employed in debugging and verifying the modem design. The primary measurement tool was an oscilloscope. For measuring timing, digital outputs were used such as LED ports or TNC port bits. For measuring signal data, the D/A channel of the DSP-93 was used to output various test points. Test programs were written that run on a PC to send and receive data over the UART link. Final system tests and tweaking were done with the following setup:



An old AMD 7910 modem chip was used to generate AFSK signals. These signals were variably attenuated and then mixed with pseudo gausian noise generated by a Texas Instruments C5x DSK board and some software. This setup was used to tweak the data clock phase lock loop, the carrier detector, and data “Eye” patterns.

The final tests were on the air tests, connecting to various local nodes.

MODEM PERFORMANCE

The data handling modes of the modem all seemed to work as planned. Not a whole lot of data was transferred using the full duplex mode except in loop back tests. The processor load was roughly measured by measuring the peak and average time it took to service all seven software modules in the main code loop. The minimum time around the loop was 17 uSec. Peak processing time around the loop while performing full duplex loopback testing, was around 55 uSec. This means the Sample_Que probably never gets even one sample behind. The average time around the loop was about 35 uSec. This means the processor is running about 38% of a full load at the present sample period of 91.8uSec. Code size takes about 2.3K of program space.

Areas for Improvement:

- The modem is not very sensitive. The carrier detect is more sensitive than the data recovery. (This is not all bad since it prevents some channel interference by keeping the transmitter off when carrier is detected.) The phase delay discriminator is probably not the ideal detection method. Also the clock recovery uses data edges as a reference which is not the ideal. A better way would probably be to mix the incoming signal to baseband and integrate and dump all the I and Q signals for one bit time. Clock recovery could use early-late integrators dumped at half bit times to generate a clock error signal.

- The Carrier detect seems to lock on to certain noise bursts. Two different receivers were tested and they both exhibited the same phenomena. Maybe some mix of RSSI and clock presence could be used to qualify the signal.

- Perhaps some form of AGC could be implemented to give a wider dynamic range on the input without the need for tweaking the DSP-93 pots.

- There is no watchdog timer in the DSP-93 to prevent a stuck PTT signal which could occur if the program got lost due to power fluctuations, RF interference, code instabilities [quite likely :>)] or other random occurrences in the universe. It has not happened during testing, but one should be careful with long unattended operation.

Ideas for the Future:

- Since only 38% of the processor is used, perhaps a PSK or higher speed modem could be added to the basic structure. There is plenty of code space left so other modems could be selected using the KISS Hardware parameter byte.

- Perhaps someone could implement full featured TNC functionality. This probably doesn't make a lot of sense given the amount of assembly code involved. Perhaps if a cheap "C" compiler becomes available…

References

- Frank H. Perkins, Jr. “Experimental 1200bps AFSK modem” DSP-93 code
- David L. Mills, “FSK modem/TNC for HF RTTY and SITOR”
- Mike Chepponis K3MC, Phil Karn, KA9Q “The KISS TNC: A simple Host-to-TNC communications protocol”
- ARRL, “AX.25 Amateur Packet-Radio Link_Layer Protocol” Ver.2.0
- Marvin E. Frerking, “Digital Signal Processing in Communication Systems”
- Lavell Jordan. “Simplified Digital Signal Processing”
- IBM, “IBM Synchronous Data Link Control General Information”
- Texas Instruments, “TMS320C2x Users Guide”
- Texas Instruments, “Digital Signal Processing Applications with the TMS320 Family Theory, Algorithms, and Implementations” Vol. 2